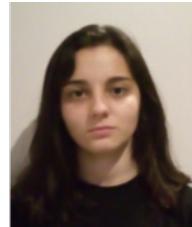


Universidade do Minho

Mestrado em Engenharia Informática

Visualização em Tempo Real

Terrain Generation



Joana Dantas
pg47288



Cláudio Moreira
pg47844



Francisco Peixoto
pg47194

Junho, 2023

1 Introdução

Neste trabalho realizado no âmbito da UC de Visualização em Tempo Real, foi proposta a escolha de um tema para um trabalho de pesquisa e análise, o grupo escolheu Terrain Generation não só por ser um tema interessante, mas também pela componente visual e interativa que iremos abordar em detalhe mais à frente.

Deste modo, o presente relatório pretende expor todo o trabalho efetuado, detalhando todos os passos que compuseram a sua realização e justificações das decisões tomadas pelo grupo.

A primeira etapa do trabalho consistiu na análise de vários algoritmos de geração de terrenos, e uma respetiva comparação baseada em critérios como complexidade e qualidade do terreno gerado.

2 Escolha do Algoritmo

Na escolha do algoritmo para a realização do trabalho tivemos em conta vários métodos de geração de terreno como:

Algoritmo Diamond-Square: O algoritmo de Diamond-Square consiste numa técnica de geração de terrenos através da subdivisão aleatória do terreno, este algoritmo é bastante popular por diversas razões, entre elas a simplicidade, rapidez e variedade.

Algoritmos Baseados em Erosão: Os algoritmos baseados em erosão consistem em simular processos naturais de erosão e sedimentação de forma a gerar paisagens realistas. Estes algoritmos têm bastantes vantagens quer a nível do realismo, quer a nível da variedade dos terrenos e na diversidade dos processos, no entanto apresentam algumas desvantagens a nível da complexidade computacional, que introduz bastante tempo de processamento e dificuldades na implementação.

Perlin Noise e Simplex Noise: Os algoritmos baseados em Perlin Noise e Simplex Noise são métodos de geração de ruído que são utilizados para gerar texturas. Perlin Noise gera um ruído suave, que quando aplicado a um mapa de altura, resulta em terrenos com colinas e vales suaves. Já o Simplex Noise corrige algumas limitações do Perlin Noise, como a sua tendência para apresentar alinhamento visual em certos ângulos, e é computacionalmente mais eficiente em altas dimensões. Estes dois algoritmos apresentam bastantes vantagens, já que podem ser bastante flexíveis e ajustáveis de acordo com as paisagens pretendidas, ambos os algoritmos têm também uma performance boa já que são eficientes em termos de complexidade, mas apresentam ser menos realistas do que as alternativas mencionadas anteriormente.

Visto as características dos diferentes algoritmos, e quando comparando as diferentes implementações dos mesmos, o grupo decidiu abordar, numa primeira implementação, o algoritmo Diamond-Square para a geração de terrenos, focando principalmente pela sua eficiência e simplicidade, que mesmo assim resultam em paisagens realistas e variadas.

2.1 1ª Implementação

Para a implementação do algoritmo Diamond-Square utilizamos o **Python**, para facilitar com a visualização e geração dos resultados, que iram ser abordados posteriormente, aproveitando as bibliotecas numpy, matplotlib e scipy para manipulação de matrizes e visualização de dados.

Para a geração do mapa de alturas criamos algumas constantes, bem como array que representa o mapa.

```
1 EDGE_SIZE = 1 + 2 ** 64    # Edge size of the resulting image in pixels
2 ROUGHNESS_DELTA = 0.7      # Roughness delta, 0 < ds < 1
3 PERIODIC = True
4 heightmap = np.zeros((EDGE_SIZE, EDGE_SIZE))
```

A função **calculate_average_fixed** calcula a média dos pontos vizinhos para os limites fixos.

Isto é, se o ponto estiver fora do heightmap, ele é ignorado.

```

1 def calculate_average_fixed(d, i, j, v, offsets):
2     """Calculate average for fixed boundaries."""
3     n = d.shape[0]
4
5     total, count = 0, 0
6     for p, q in offsets:
7         pp, qq = i + p * v, j + q * v
8         if 0 <= pp < n and 0 <= qq < n:
9             total += d[pp, qq]
10            count += 1.0
11    return total / count

```

A função *calculate_average_periodic*, por outro lado, calcula a média dos pontos vizinhos para limites periódicos, ou seja, considera o heightmap como se ele fosse cíclico.

```

1 def calculate_average_periodic(d, i, j, v, offsets):
2     """Calculate average for periodic boundaries."""
3     n = d.shape[0] - 1
4
5     total = 0
6     for p, q in offsets:
7         total += d[(i + p * v) % n, (j + q * v) % n]
8     return total / 4.0

```

A função *diamond_square_step* é a implementação base do algoritmo, que calcula a média dos pontos vizinhos e adiciona um deslocamento aleatório. [7]

```

1 def diamond_square_step(heightmap, size, roughness_delta, avg_func):
2     """Apply diamond square step."""
3     n = heightmap.shape[0]
4     half_size = size // 2
5
6     diamond_offsets = [(-1, -1), (-1, 1), (1, 1), (1, -1)]
7     square_offsets = [(-1, 0), (0, -1), (1, 0), (0, 1)]
8
9     # Diamond Step
10    for i in range(half_size, n, size):
11        for j in range(half_size, n, size):
12            heightmap[i, j] = avg_func(heightmap, i, j, half_size, diamond_offsets)
13            + random.uniform(-roughness_delta,
14                               roughness_delta)
15
16    # Square Step, rows
17    for i in range(half_size, n, size):
18        for j in range(0, n, size):
19            heightmap[i, j] = avg_func(heightmap, i, j, half_size, square_offsets)
20            + random.uniform(-roughness_delta,
21                               roughness_delta)
22
23    # Square Step, cols
24    for i in range(0, n, size):
25        for j in range(half_size, n, size):
26            heightmap[i, j] = avg_func(heightmap, i, j, half_size, square_offsets)
27            + random.uniform(-roughness_delta,
28                               roughness_delta),

```

O algoritmo itera em várias etapas, denominadas como Diamond Step e Square Step.

Na Diamond Step itera-se sobre cada pixel no heightmap em intervalos da variável size e adicionamos um deslocamento aleatório à média dos quatro pontos de canto, o que cria um efeito de rugosidade.

Na Square Step itera-se sobre cada linha e coluna, novamente adicionando um deslocamento aleatório à média dos quatro pontos vizinhos.

Por fim, a função *generate_heightmap* executa o algoritmo Diamond-Square em várias escalas, criando fractais de terreno em múltiplas resoluções. Começamos com o tamanho completo do

heightmap e dividimos o tamanho por dois em cada iteração, ao mesmo tempo em que aumentamos a rugosidade, até que o tamanho seja 1.

```

1 def generate_heightmap(heightmap, edge_size, roughness_delta, is_periodic):
2     """Generate heightmap using the Diamond-Square algorithm."""
3     size, roughness = edge_size-1, 1.0
4     avg_func = calculate_average_periodic if is_periodic else
5     calculate_average_fixed
6     while size > 1:
7         diamond_square_step(heightmap, size, roughness, avg_func)
8
9         size /= 2
10        roughness *= roughness_delta
11
12    return heightmap

```

2.2 Análise de Complexidade da Implementação

A complexidade do algoritmo, em termos de tempo, é determinada pela função *diamond_square_step* que é chamada para cada nível de detalhe do heightmap. Em cada chamada desta função, dois loops são executados sobre as dimensões do heightmap, resultando portanto numa complexidade de $O(n^2)$ para cada chamada, onde n é a dimensão do heightmap (EDGE_SIZE).

Já em termos de espaço, como o algoritmo guarda o heightmap, a sua complexidade também é $O(n^2)$.

Também é importante mencionar que a Rugosidade (ROUGHNESS_DELTA) afeta a amplitude dos deslocamentos aleatórios aplicados durante o algoritmo, que apesar de não terem um impacto grande na complexidade do algoritmo, podem influenciar bastante a performance em termos do que se espera do resultado final.

2.3 Comparação com outros Algoritmos

Para escolha do algoritmo final, realizou-se uma análise entre as diferentes possibilidades. Essa análise levou em consideração vários aspectos importantes, como o impacto na CPU e GPU, os FPS médios obtidos e o tempo de execução como é possível observar através das seguintes imagens:

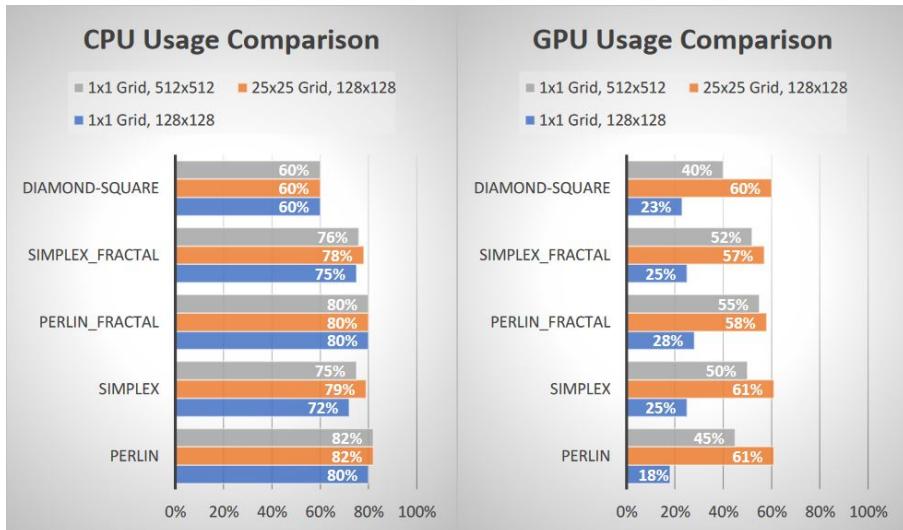


Figura 1: Comparação entre os algoritmos, impacto na CPU e GPU [5]

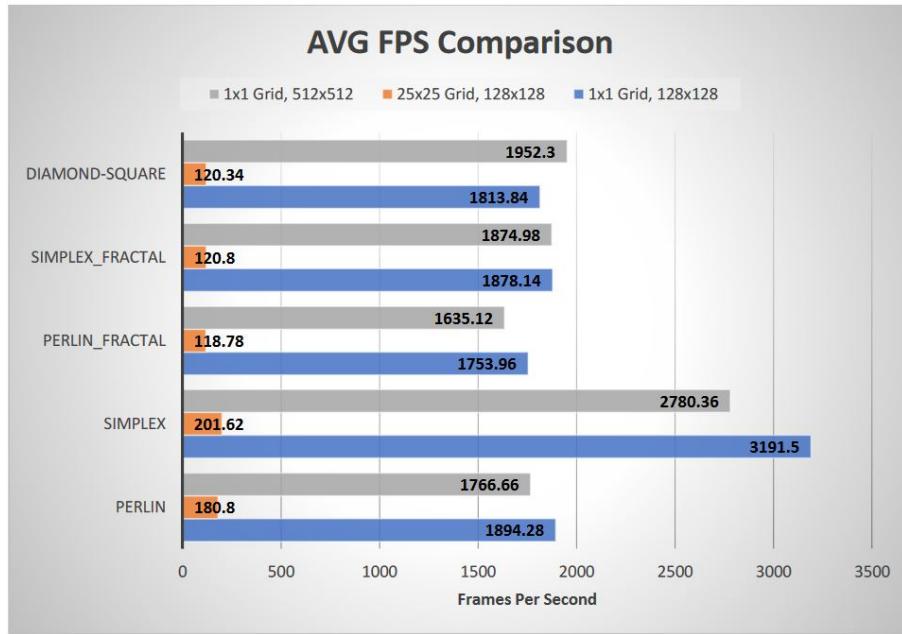


Figura 2: Comparaçāo entre os algoritmos em termos de FPS [5]



Figura 3: Comparaçāo entre os algoritmos em termos de tempo de execuçāo [5]

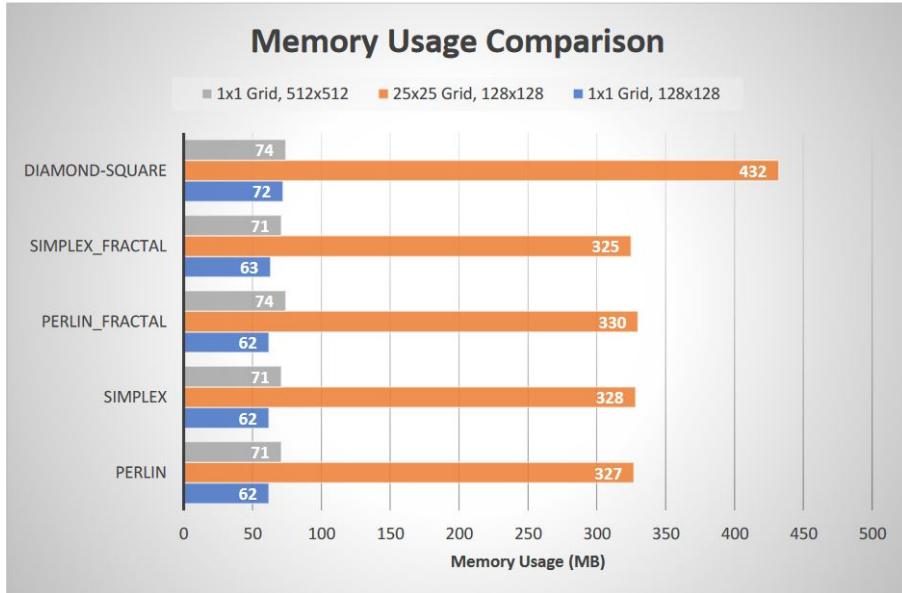


Figura 4: Comparação entre os algoritmos em termos de uso de memória [5]

Analisando os gráficos, podemos observar que a única desvantagem do algoritmo Diamond-Square perante os outros é no uso de memória, visto que a complexidade espacial é $O(n^2)$, ou seja, à medida que o tamanho do terreno cresce, aumenta também o uso de memória. No entanto, o algoritmo em estudo demonstra ser mais eficiente em termos uso de recursos. Quanto ao tempo de execução, perde apenas para o Simplex Noise.

3 Visualização do Terreno gerado

Para a visualização dos resultados, tal como mencionado anteriormente utilizamos as seguintes bibliotecas do *Python*:

```

1 import matplotlib.colors
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4 from scipy.ndimage import gaussian_filter

```

Com recurso a colormaps, conseguimos demonstrar de forma mais realista os resultados calculados pelo algoritmo.

```

1 def load_colormap(filename):
2     """Load colormap from a file."""
3     colormap = []
4     try:
5         for row in np.loadtxt(filename):
6             colormap.append([row[0], row[1:4]])
7     except FileNotFoundError:
8         print(f"File {filename} not found.")
9     return None
10 return matplotlib.colors.LinearSegmentedColormap.from_list("geo-smooth",
11                                                               colormap)

```

De forma a demonstrar o heightmap em 2D utilizamos o seguinte código:

```

1 colormap = load_colormap("geo-smooth.gpf")
2
3 if colormap is not None:
4     terrain = generate_heightmap(heightmap, EDGE_SIZE, ROUGHNESS_DELTA, PERIODIC)
5

```

```

6   plt.figure(figsize=(EDGE_SIZE / 100, EDGE_SIZE / 100), dpi=100)
7   plt.tick_params(left=False, bottom=False, labelleft=False, labelbottom=False)
8   plt.imshow(terrain, cmap=cmap)
9
10  plt.savefig("terrain.png")
11  plt.show()

```

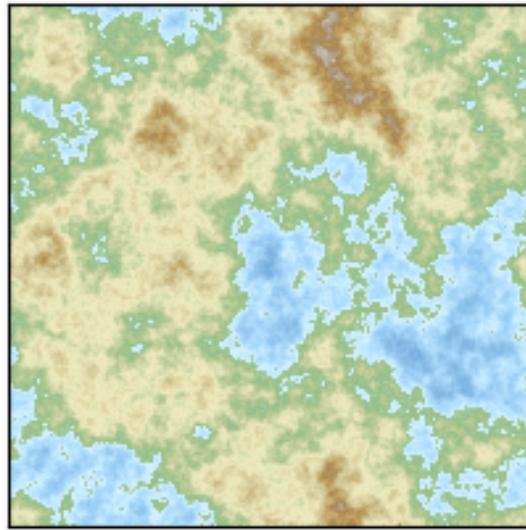


Figura 5: Mapa gerado em 2D

Utilizamos também um Plot3D para simular as alturas.

```

1 def plot_3D(heightmap, colormap):
2     """Plot the terrain in 3D."""
3     fig = plt.figure(figsize=(10, 10))
4     ax = fig.add_subplot(111, projection='3d')
5
6     x = np.arange(heightmap.shape[0])
7     y = np.arange(heightmap.shape[1])
8     X, Y = np.meshgrid(x, y)
9
10    surf = ax.plot_surface(X, Y, heightmap, cmap=colormap, linewidth=0, antialiased=False)
11
12    fig.colorbar(surf, shrink=0.5, aspect=5)
13
14    plt.show()

```

Ao corrermos pela primeira vez o nosso algoritmo, obtivemos o seguinte terreno:

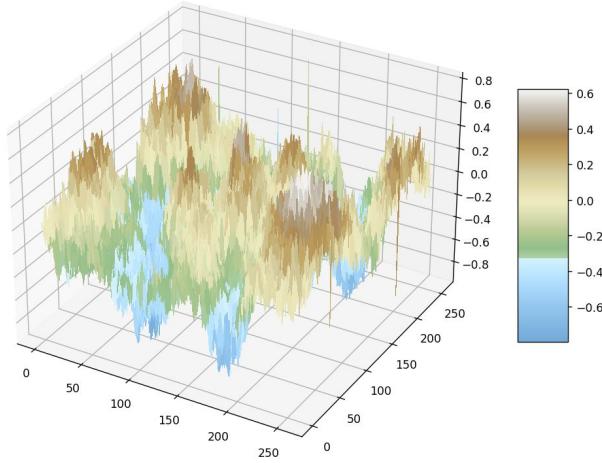


Figura 6: Primeiros resultados, com picos acentuados e elevada aleatoriedade

Ao olhar para a figura anterior, a primeira coisa que notamos é que o terreno parece bastante descontinuo. Para resolver isto, decidimos aplicar um filtro de suavização conhecido como Gaussian blur. Esta técnica consiste em aplicar uma função Gaussiana com a imagem de entrada, de forma a suavizar as descontinuidades.

Foi então necessário aplicar operações de smoothing para gerar um terreno mais uniforme e sem disparidades. Optamos por usar uma função de Gaussian filter para o efeito, e obtemos os seguintes resultados para um sigma = 5:

```

1 def smooth_heightmap(heightmap, sigma):
2     """Smooth the heightmap using a Gaussian filter."""
3     return gaussian_filter(heightmap, sigma=sigma)
4
5
6 if colormap is not None:
7     terrain = generate_heightmap(heightmap, EDGE_SIZE, ROUGHNESS_DELTA, PERIODIC)
8
9     terrain_smoothed = smooth_heightmap(terrain, sigma=1)
10
11 plot_3D(terrain_smoothed, colormap)

```

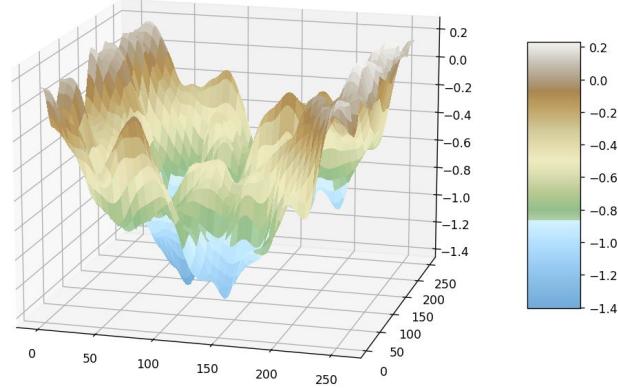


Figura 7: Resultado após aplicação de smoothing

Com a aplicação do filtro de suavização, conseguimos criar um terreno que parece mais natural e suave, com transições menos abruptas entre as diferentes alturas.

4 2^a Implementação

Para a 2^o fase do projeto, o grupo procedeu à implementação de dois algoritmos, a serem visualizados recorrendo a shaders GLSL. O motor que usamos para "montar" a pipeline 3D foi a NAU3D.

Decidimos experimentar duas abordagens diferentes: numa delas, geramos um heightmap previamente. Esse heightmap consiste numa imagem grayscale, gerada com um script em python, que será posteriormente passada para a NAU, que a converte em vértices. Após isso, colorimos o terreno com texturas apropriadas.

A outra abordagem consiste em gerar o terreno em tempo real. Ao invés de gerar um heightmap previamente, todas as operações serão feitas nos shaders.

4.1 Diamond-Square

O Diamond-Square é um bom exemplo de uma algoritmo frequentemente usado na geração de heightmaps para a geração de terrenos em videojogos [8].

Optamos por usar este algoritmo para a geração prévia de heightmaps, devidos às seguintes limitações no que toca à geração em tempo real:

- Natureza iterativa: O algoritmo Diamond-Square opera iterativamente, subdividindo o terreno e atribuindo alturas a midpoints. Este processo iterativo não é fácil de se implementar num shader, pois os shaders são concebidos para processamento paralelo e operam em vértices ou pixéis individuais de forma independente;
- Baseado em grelha: O algoritmo opera numa estrutura em grelha, onde a altura de cada ponto é determinada com base nos pontos vizinhos. Os shaders operam em vértices ou fragmentos individuais.

Começamos por implementar um script simples que usa o algoritmo Diamond-Square para gerar um heightmap em grayscale, e aplica um filtro gaussiano para aplicar smoothing sobre a imagem, de forma a obter futuramente relevos mais suaves e naturais.

De seguida, passamos esta imagem para a NAU, usando "heightmap" como uma opção para a cena no XML das configurações:

```
<scene name="Terrain">
<terrain name = "fractal" heightMap ="textures/heightmap.png" material="terrain">
<SCALE x=0.1 y=2.5 z =0.1 />
</terrain>
</scene>
```

Esta opção [4, 1] permite tomar o heightmap como input, extrair informação da altura, gerar vertex attributes (posição, normal, texture coordinates), cria triângulos baseados nos dados do heightmap, e atribui a informação resultante ao objeto "Terrain" para ser renderizado. Assim, o nosso vertex shader é bastante simples, visto que as operações que calculam a posição dos vértices estão feitas:

```
1 #version 440
2
3
4 uniform mat4 projectionViewModel;
5 uniform mat3 normalMatrix;
6 uniform vec3 lightDirection;
7 uniform mat4 viewMatrix;
8 uniform mat4 viewModel;
```

```

9   in vec4 position;
10  in vec3 normal;
11  in vec2 texCoord0;
12
13  out vec4 eye;
14  out vec2 tc;
15  out vec3 fragNormal;
16  out vec3 fragPosition;
17  out vec3 fragLightDirection;
18
19 void main() {
20     gl_Position = projectionViewModel * position;
21
22     eye = -(viewModel * position);
23     tc = texCoord0;
24     fragNormal = normalize(normalMatrix * normal);
25     fragPosition = vec3(position);
26
27     vec4 lightDirViewSpace = viewMatrix * vec4(lightDirection, 0.0);
28     fragLightDirection = normalize(-lightDirViewSpace.xyz);
29 }

```

O terreno que vamos gerar vai ser do tipo tundra. Portanto, optamos por usar uma textura de vegetação rasteira e outra textura de gelo/neve. Para ajustar a altura a partir da qual se observa neve, definimos um valor "snowThreshold" no fragment shader:

```

1 #version 440
2
3 uniform sampler2D terrainTexture;
4 uniform sampler2D snowTexture;
5
6 in vec4 eye;
7 in vec2 tc;
8 in vec3 fragNormal;
9 in vec3 fragPosition;
10 in vec3 fragLightDirection;
11
12 out vec4 fragmentColor;
13
14 void main() {
15     // Determine the height threshold for snow
16     float snowThreshold = 1.9; // Adjust this value to control the snow coverage
17
18     // Sample the grass texture
19     vec4 grassColor = texture(terrainTexture, tc);
20
21     // Sample the snow texture
22     vec4 snowColor = texture(snowTexture, tc);
23
24     // Calculate lighting intensity
25     vec3 surfaceNormal = normalize(fragNormal);
26     float lightIntensity = dot(surfaceNormal, fragLightDirection);
27
28     // Check if the height of the terrain is above the snow threshold
29     if (fragPosition.y >= snowThreshold) {
30         // Use snow texture for high terrain
31         fragmentColor = vec4(snowColor.rgb * lightIntensity, snowColor.a);
32     } else {
33         // Use grass texture for low terrain
34         fragmentColor = vec4(grassColor.rgb * lightIntensity, grassColor.a);
35     }
36 }

```

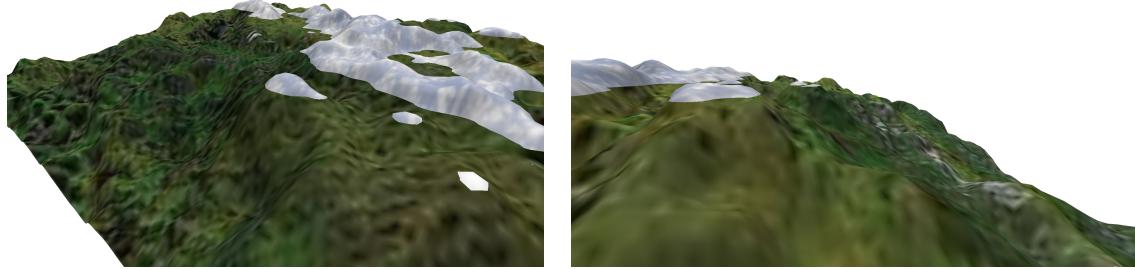


Figura 8: Terreno gerado com Diamond-Square, para roughness=0.7 e sigma(gaussian smoothing)=1.0 (perspectivas diferentes)

4.2 SimplexNoise

Para a implementação em tempo real, as escolhas mais populares são os algoritmos Perlin Noise e Simplex Noise.

Em 2001, Ken Perlin, o autor do Perlin Noise, apresentou o Simplex Noise como uma alternativa que resolvia os problemas do Classic "Perlin" Noise. [6]

O algoritmo clássico usa hipercubos: em 2D é um quadrado, em 3D é um cubo, etc. O problema é o facto da interpolação linear apenas funcionar em 1 dimensão. Por exemplo, para interpolar 4 pontos, como no Perlin Noise 2D, 2 lados paralelos devem ser interpolados, e depois esses 2 resultados tem de ser interpolados também. Isto pode ser equacionado como $2^n - 1$ interpolações lineares, que em 2D são 3, em 3D são 7 e 4D são 15.

Em vez de hipercubos, o algoritmo Simplex usa o simplex: em 2D é um triângulo, em 3D é um tetraedro, etc. A ideia é que em vez de termos um hipercubo de N dimensões que tem 2^N cantos, temos um simplex de N dimensões, que tem apenas $N + 1$ cantos.

Além disso, um problema fundamental do ruído clássico Perlin é que ele envolve interpolações sequenciais ao longo de cada dimensão. Além do rápido aumento na complexidade computacional à medida que avançamos para dimensões mais altas como falamos anteriormente, torna-se cada vez mais difícil calcular a derivada analítica da função interpolada. O ruído simplex, por outro lado, utiliza uma soma direta das contribuições de cada canto, onde a contribuição é uma multiplicação da extração do gradiente e de uma função de atenuação radialmente simétrica. A atenuação radial é cuidadosamente escolhida para que a influência de cada canto atinja zero antes de cruzar a fronteira para o próximo simplex. Isso significa que os pontos dentro de um simplex serão influenciados apenas pelas contribuições dos cantos desse simplex em particular.

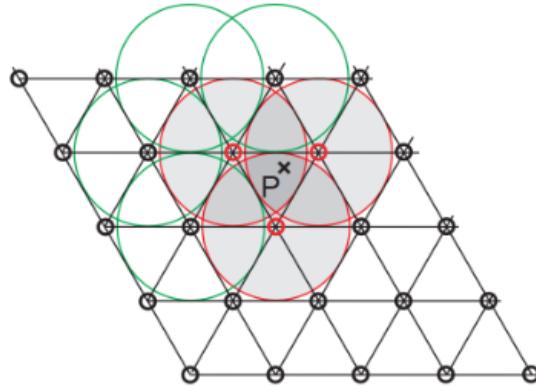


Figura 9: Um ponto P dentro de um simplex recebe contribuições para o seu valor apenas a partir dos três núcleos centrados nos cantos circundantes (marcados por círculos vermelhos e sombreados). Os núcleos nos cantos mais distantes (círculos verdes) decaem a zero antes de atravessar a fronteira para o simplex que contém P. Assim, o valor do ruído em cada ponto pode sempre ser calculado como a soma de três termos.

Em suma, o grupo escolheu o algoritmo Simplex Noise pelas seguintes razões:

- O Simplex Noise tem menor complexidade computacional e requer menos multiplicações que o Perlin Noise;
- Em termos de dimensões, a complexidade do Simplex é $O(N^2)$ em vez de $O(2^N)$ do Perlin Noise;
- O Simplex Noise não gera artefactos direcionais que se notem;
- O Simplex Noise tem um gradiente bem definido e contínuo, o que se traduz em transições mais suaves entre valores e consequentemente a geração de terrenos de relevo mais natural.

Assim sendo, incorporamos o Simplex Noise para o cálculo em tempo real da altura dos pixels. Para o efeito, usamos uma função de Simplex Noise em 3D [3], por sua vez adaptada da implementação de Stefan Gustavson [6, 2]. Em termos de textura e iluminação, usamos a mesma abordagem que na implementação do terreno com heightmaps descrita anteriormente, pelo que o fragment shader é praticamente igual. É no vertex shader que são feitos os cálculos que atribuem uma altura aos fragmentos através da função de noise:

```

1 #version 440
2
3
4 uniform mat4 projectionViewModel;
5 uniform mat3 normalMatrix;
6 uniform vec3 lightDirection;
7 uniform mat4 viewMatrix;
8 uniform mat4 viewModel;
9
10 in vec4 position;
11 in vec3 normal;
12 in vec2 texCoord0;
13
14 out vec4 eye;
15 out vec2 tc;
16 out vec3 fragNormal;
17 out vec3 fragPosition;

```

```

18 out vec3 fragLightDirection;
19
20
21 vec4 _permute(vec4 x) { return mod(((x * 34.0) + 1.0) * x, 289.0); }
22 vec4 _taylorInvSqrt(vec4 r) { return 1.79284291400159 - 0.85373472095314 * r; }
23
24 // Simplex noise function from here: https://github.com/FarazzShaikh/glNoise/blob/master/src/Simplex.glsl
25 float gln_simplex(vec3 v) {
26     const vec2 C = vec2(1.0 / 6.0, 1.0 / 3.0);
27     const vec4 D = vec4(0.0, 0.5, 1.0, 2.0);
28
29     // First corner
30     vec3 i = floor(v + dot(v, C.yyy));
31     vec3 x0 = v - i + dot(i, C.xxx);
32
33     // Other corners
34     vec3 g = step(x0.yzx, x0.xyz);
35     vec3 l = 1.0 - g;
36     vec3 i1 = min(g.xyz, l.zxy);
37     vec3 i2 = max(g.xyz, l.zxy);
38
39     // x0 = x0 - 0. + 0.0 * C
40     vec3 x1 = x0 - i1 + 1.0 * C.xxx;
41     vec3 x2 = x0 - i2 + 2.0 * C.xxx;
42     vec3 x3 = x0 - 1. + 3.0 * C.xxx;
43
44     // Permutations
45     i = mod(i, 289.0);
46     vec4 p = _permute(_permute(_permute(i.z + vec4(0.0, i1.z, i2.z, 1.0)) + i.y +
47                             vec4(0.0, i1.y, i2.y, 1.0)) +
48                             i.x + vec4(0.0, i1.x, i2.x, 1.0));
49
50     // Gradients
51     // ( N*N points uniformly over a square, mapped onto an octahedron.)
52     float n_ = 1.0 / 7.0; // N=7
53     vec3 ns = n_ * D.wyz - D.xzx;
54
55     vec4 j = p - 49.0 * floor(p * ns.z * ns.z); // mod(p,N*N)
56
57     vec4 x_ = floor(j * ns.z);
58     vec4 y_ = floor(j - 7.0 * x_); // mod(j,N)
59
60     vec4 x = x_ * ns.x + ns.yyyy;
61     vec4 y = y_ * ns.x + ns.yyyy;
62     vec4 h = 1.0 - abs(x) - abs(y);
63
64     vec4 b0 = vec4(x.xy, y.xy);
65     vec4 b1 = vec4(x.zw, y.zw);
66
67     vec4 s0 = floor(b0) * 2.0 + 1.0;
68     vec4 s1 = floor(b1) * 2.0 + 1.0;
69     vec4 sh = -step(h, vec4(0.0));
70
71     vec4 a0 = b0.xzyw + s0.xzyw * sh.xxxy;
72     vec4 a1 = b1.xzyw + s1.xzyw * sh.zzww;
73
74     vec3 p0 = vec3(a0.xy, h.x);
75     vec3 p1 = vec3(a0.zw, h.y);
76     vec3 p2 = vec3(a1.xy, h.z);
77     vec3 p3 = vec3(a1.zw, h.w);
78
79     // Normalise gradients
80     vec4 norm =
81         _taylorInvSqrt(vec4(dot(p0, p0), dot(p1, p1), dot(p2, p2), dot(p3, p3)));
82     p0 *= norm.x;
83     p1 *= norm.y;

```

```

84     p2 *= norm.z;
85     p3 *= norm.w;
86
87     // Mix final noise value
88     vec4 m =
89         max(0.6 - vec4(dot(x0, x0), dot(x1, x1), dot(x2, x2), dot(x3, x3)), 0.0);
90     m = m * m;
91     return 42.0 *
92         dot(m * m, vec4(dot(p0, x0), dot(p1, x1), dot(p2, x2), dot(p3, x3)));
93 }
94
95
96 void main() {
97     eye = -(viewModel * position);
98     tc = texCoord0;
99
100    // Generate height using simplex noise
101    float scale = 0.03; // Adjust the scale factor to control the height variations
102    float offset = 0.5; // Adjust the offset to control the baseline height
103    float height = (gln_simplex(position.xyz * scale) + offset) * 10.0; // Adjust
104    // the multiplier for desired mountain height
105
106    // Apply the generated height to the vertex position in object space
107    vec4 modifiedPosition = position;
108    modifiedPosition.y += height; // Add the generated height
109
110    // Transform the modified position to eye space
111    //eye = viewMatrix * modifiedPosition;
112    gl_Position = projectionViewModel * modifiedPosition;
113
114    // Pass the modified position to the fragment shader
115    fragPosition = vec3(modifiedPosition);
116
117    // Calculate lighting
118    fragNormal = normalize(normalMatrix * normal);
119    vec4 lightDirViewSpace = viewMatrix * vec4(lightDirection, 0.0);
120    fragLightDirection = normalize(-lightDirViewSpace.xyz);
121 }

```

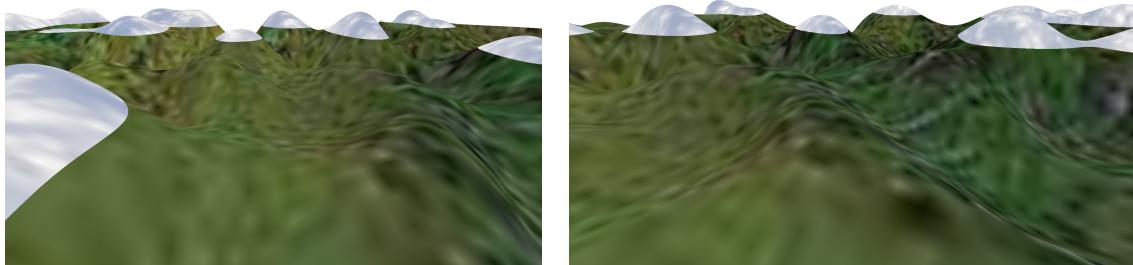


Figura 10: Resultados da nossa implementação com o algoritmo Simplex

5 Conclusão

Através deste projeto, o grupo teve a oportunidade de explorar vários métodos de geração de terreno.

A implementação do algoritmo, bem como a posterior visualização do terreno gerado, permitiu-nos perceber a importância de cada passo, desde a geração inicial do mapa de alturas, passando pela aplicação do algoritmo, até à representação final do terreno, bem como entender como melhorar os resultados obtidos.

A comparação com outros algoritmos ajudou-nos a entender as vantagens e desvantagens de

cada um, permitindo uma escolha informada. Aquando da 1º implementação, o Diamond-Square revelou-se um bom compromisso entre a eficiência, os recursos computacionais e a qualidade dos resultados. Por fim, a suavização do terreno através do filtro de suavização permitiu-nos aprimorar ainda mais a qualidade visual do terreno gerado.

Na fase de implementação em shaders, a escolha de duas abordagens diferentes permitiu-nos explorar as vantagens e desvantagens de cada método, bem como perceber as possíveis aplicações práticas. O uso do algoritmo Diamond Square para a geração prévia de heightmaps revelou ser útil para se conseguir produzir paisagens com detalhes mais destacados, como vales profundos ou relevos acentuados e irregulares, bem como controlar melhor o aspetto do terreno ajustando os parâmetros do algoritmo. Já o algoritmo que usa Simplex Noise revelou ser a melhor escolha para a geração em tempo real, que produz terrenos com contornos mais suaves e regulares.

Por fim, a suavização do terreno através do filtro de suavização permitiu-nos aprimorar ainda mais a qualidade visual do terreno gerado.

No futuro, poderíamos ainda implementar melhorias, como texturas ou a implementação de características específicas do terreno, como por exemplo a formação de rios ou a simulação de diferentes tipos de vegetação.

Referências

- [1] Commit: Added heightmap terrain as a scene option, <https://github.com/nau3d/>.
- [2] noise3d.gsls, <https://github.com/ashima/>.
- [3] Simplex.gsls, <https://github.com/farazzshaikh>.
- [4] "terrain", documentação da nau3d.
- [5] Bailey Gardner. Analysis of the performance and effectiveness of techniques for real-time terrain generation. 2019.
- [6] Stefan Gustavson. Simplex noise demystified. 01 2005.
- [7] Phillip Janert. The diamond-square algorithm for terrain generation.
- [8] Thomas J. Rose and Anastasios G. Bakaoukas. Algorithms and approaches for procedural terrain generation - a brief review of current techniques. In *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–2, 2016.